Traceable Recursion with Graphical Illustration for Novice Programmers

Leonardo Sa
Student, Department of Computer Science, Information Systems, and Mathematics
Park University

Wen-Jung Hsin, PhD
Professor, Department of Computer Science, Information Systems, and Mathematics
Park University

Recursion is a concept that can be used to describe the phenomena and natural occurrences in many different fields. As many applications utilize computer software to model recursion, recursion is a particularly important concept in the computing discipline. However, it is a difficult concept for many undergraduate students to master. A Recursion Graph (RGraph) is one visualization method for representing recursion. This paper extends our previous work on RGraphs to include a tool for automatically generating complete and partial RGraphs from an arbitrary recursive program. Use of this tool allows for more flexibility in demonstrations and more focused pedagogical interactions on the part of students, thereby improving student learning in recursion.

In mathematics, recursion is a method of defining a mathematical function based on previously defined terms of the same function. It is an important concept in Computer Science as well as many other disciplines. McCracken (1987) stated that "Recursion is fundamental in Computer Science, whether understood as a mathematical concept, a programming technique, a way of expressing an algorithm, or a problem-solving approach." In computing, it appears frequently in the study of algorithms, data structures, and artificial intelligence. In other fields of study, recursion appears as population and predator/prey models in biology, formal structures in linguistics, filters in signal processing, and genomic sequencing in bioinformatics. Fractals are self-similar, recursive patterns found in nature and simulated through mathematics with applications in art, design, and engineering.

Although recursion is an important concept, teaching recursion to introductory Computer Science students is a challenging task (AP Central.) This has been documented in several studies. For example, AP Central states that "It is not uncommon for novice programmers to have difficulty understanding recursion," and Dann, Cooper, and Pausch (2000) argued, and Gal-Ezer and Harel (1998) agreed, that "Some Computer Science educators have described the process of teaching recursion as one of the universally most difficult concepts to teach." Teaching recursion is a challenge largely because students have a difficult time envisioning the abstract concept. A variety of approaches and studies has been tried to better explain this concept. For example, Wu, Dale, and Bethel (1998) used experiment results to show that the concrete conceptual models are better than abstract conceptual models. Hundhausen, Douglas, and Stasko (2002) show that student use of algorithm visualization technology has a great impact on teaching effectiveness. The experiment result in Bruce, Danyluk, and Murtagh (2005) shows that presenting recursive structure to students earlier rather than later can help reinforce the concept of recursion and better prepare students for other data structures. As many varieties of applications in other fields such as signal processing, human genomic sequencing, and population modeling have utilized computer software to capture the idea of recursion, it is important that Computer

Science students understand the concept of recursion early on so that they are prepared to write programs to capture the recursive phenomena and natural occurrences in many different applications.

With the suggestion of the work from Hundhausen et. al. (2002) and Wu et. al. (1998) where concrete conceptual modeling and visualization technology enhance teaching effectiveness, this paper discusses the Recursion Graph (RGraph) to try to help novice programmers to learn the concept of recursion. RGraph was developed in 1996 and initially documented in Hsin (2008). In 2009, we implemented a software tool to automatically generate complete and partial RGraphs. A complete RGraph provides a concrete conceptual modeling tool that can help crystallize for students the concept of recursion. A partial RGraph can help student learning by having students think about what labels are missing, thereby assessing their understanding of the concept. We report on the experimental result of student learning in using RGraphs.

One particular feature of an RGraph is that it is traceable. Specifically, it shows the detailed invocation sequences from one layer to another such that the flow of the calls is traceable. Since an RGraph is traceable, it can be used as a self debugging tool. It is particularly useful when the department of Computer Science at the university where the authors teach adopted PDProlog (Public-Domain Prolog) fourteen years ago. At that time, PDProlog was the only free Prolog interpreter for use in the personal computer. PDProlog, however, did not provide a trace command for the purpose of debugging. The authors therefore invented RGraph, providing the needed debugging tool. When a student wishes to see how his recursion algorithm works, he is required to hand draw an RGraph, starting from the simplest case (such as N = 1 or 2). This helps the student catch his own mistakes if the algorithm has problems.

Our investigations can help Computer Science educators better understand how students learn a critical and difficult concept in the discipline by using algorithm visualization technology.

In the literature, many books (Cormen, Leiserson, Riverst, and Stein, C., 2009; Horstmann, 2002) and lecture notes posted on the Internet (National Institute of Standards and Technology, 2005; Recursion Tree, 2007; Recursion Tree,1997; Turbak, 2001) use recursion trees showing how recursion progresses to degenerated cases. The recursion trees in these references indicate the abstract algorithmic recurrence relationship. Our RGraph shows the detailed invocation sequences from one layer to another, such that the flow of the calls is traceable. The precise difference between a recursion tree and an RGraph will be discussed in the section entitled "Comparison between an RGraph and a Recursion Tree." Additionally, various algorithm animations (Davidson, n.d.; Jeliot, n.d.; JHAVEPOP, n.d.; McHugh, n.d.; Haug, n.d.; Stern and Naish, 2002) are also available on the Internet. Our RGraph software tool differs from these animations in that the flow of calling sequence is depicted in RGraphs, such that one can trace the process of recursion explicitly.

This paper performs pedagogical investigations into a technique to improve student learning in Computer Science education. The focus on student learning is one of the key elements explored by Scholarship of Teaching and Learning (SoTL) (Bruff, n,d.; Hutchings & Shulman, 1999). Hutchings and Shulman (1999) state that "SoTL is not only done publicly to invite critical review and exchange of ideas but also with an emphasis on inquiry into student learning." Our investigations can help Computer Science educators better understand how students learn a critical and difficult concept in the discipline by using algorithm visualization technology.

The rest of the paper is organized as follows. The "Teaching the RGraph" section defines an RGraph, and describes the functionalities of RGraph software

tool. The "RGraph Examples" section provides several examples of constructing RGraphs. The "Comparison between an RGraph and a Recursion Tree" section compares an RGraph with a recursion tree. The "Experimental Result in Student Learning" section reports the experimental result of student learning in using RGraphs.

Teaching the RGraph

In the sections that follow, we define RGraphs and describe how students are introduced to RGraphs. To help readers understand how RGraphs help students learn the concept of recursion, three examples are introduced in the main paper. The first two examples, forward ( ) and backward ( ) functions, are two of the most revealing examples in demonstrating the concept of recursion in the authors' teaching experience. In particular, using these two examples, beginning students can trace the flow of recursion, grasp the elements involved in recursion (i.e., terminating condition, n^{th} term depending on $(n-1)^{th}$ term), and understand the importance of the placement of a recursion call in the program. The third example illustrates how RGraph shows the process of recursion more clearly as compared to the common recursion tree approach in the current literature. The experimental result following these sections shows how student learning is improved in understanding the concept of recursion.

Definition

An RGraph is a directed graph, showing the invocation sequence of function calls. It is built layer by layer from top to bottom (i.e., breadth-first instead of depth-first), with directed edges indicating the processing sequence. To trace a recursion algorithm in an RGraph, depth-first search is used. Except for the directed cycles formed by the edges, an RGraph looks a lot like a tree.

Formally, an RGraph is a directed graph consisting of a set of vertices, V, and a set of directed edges, E. There are two types of vertices in set V: oval and square. An oval vertex indicates a recursion call, whereas a square vertex shows a pre-processing statement prior to a recursion call or a post-processing statement after a recursion call.

A vertex can have multiple outgoing edges, pointing to different directions: (1) down to a vertex in the next lower layer, (2) right to a vertex in the same layer, or (3) up to a vertex in the next higher layer. The order of the execution sequence is (1), (2), and (3) for any existent outgoing edges. In essence, depth-first search is observed. More precisely, if a vertex has a downward pointing edge, the vertex pointed by the edge will be executed first. The upward pointed vertex will be called last after the current vertex has been executed.

RGraph Software

The RGraph software tool was designed and implemented at the university where the authors teach in early 2009. Its user interface is shown in Figure 1, in which a user can specify a computer program and the methods within the program to be traced. After the user clicks on the "Generate Graphs!" button, a graphical output is generated showing the sequence of method calls. For graphical accessibility, a user can zoom in and out of a graphical output display.

RGraph Examples

In this section, examples are shown using a prototype language similar to the syntax in Java programming language. Note that an exact programming language is not used in this paper, simply because many programming language such as C, C++, Java, Prolog, and Lisp can be used to implement the algorithms.

Example: Recursive Print

Printing a list of elements in a forward order or a backward (i.e., reverse) order can be done using a recursive algorithm. The following show both forward and backward printing algorithms

```
forward(LIST) {
        if (LIST == empty)
                return;
        else {
                print(head(LIST));
                forward(tail(LIST));
        }
}
backward(LIST) {
         if (LIST == empty)
                return;
        else {
                backward(tail(LIST));
                print(head(LIST));
        }
}
```

where function head(LIST) extracts the first element in the LIST, and function tail(LIST) returns a list consisting of the rest of the elements excluding the head element. For example, head(ABCDE) returns element A, and tail(ABCDE) returns the list BCDE. Notice that the difference between forward( ) and backward( ) is simply the position of print( ) function relative to the recursive invocation.

Figure 2 shows an RGraph for printing list ABCDE by invoking forward("ABCDE"). Figure 3 shows an RGraph for backward("ABCDE"). Notice that a vertex such as forward("BCDE") in Figure 2 has multiple outgoing edges. In this case, the edge going downward to the lower layer should be executed first, effectively, performing the lower layer subroutine call first. By following the sequence of print( ) statements in both Figures 2 and 3, one can obtain the printed orders for list ABCDE.

Example: Partial RGraph

The field "Percentage of missing labels" in Figure 1 indicates whether an RGraph is complete (i.e., 0% missing label), or partial. Figure 4 shows an example of an RGraph where 30% of labels are missing from the complete RGraph in Figure 2. A partial RGraph can assist student learning by having students think about what labels are missing, and can be used to assess students' understanding of recursion concept.

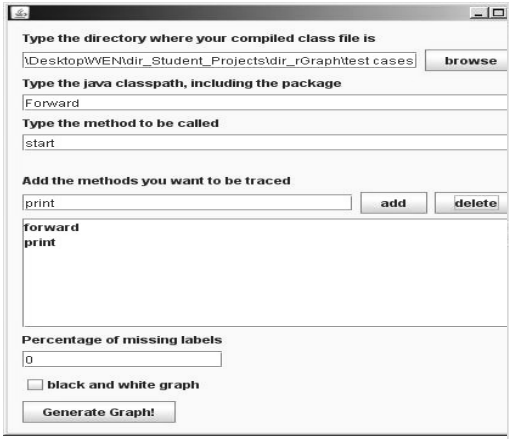

Figure 1. RGraph Software Tool – User Interface Panel

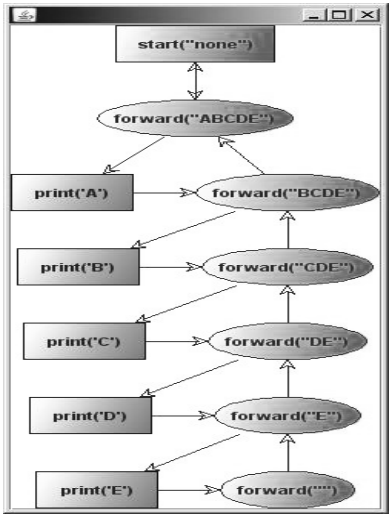

Figure 2. An RGraph for Forward Printing of list "ABCDE"

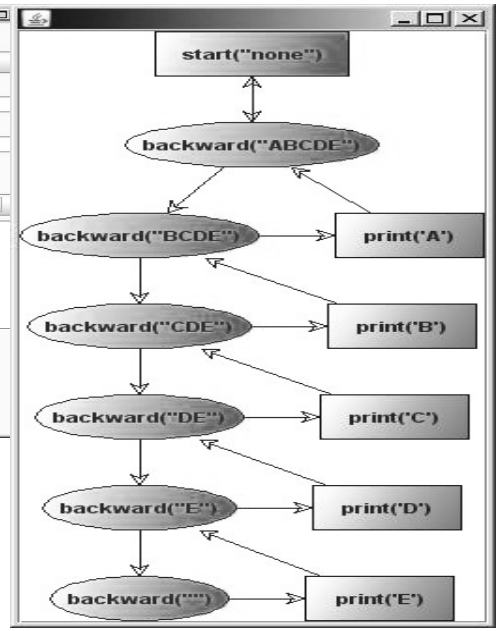

Figure 3. An RGraph for Backward Printing of list "ABCDE"

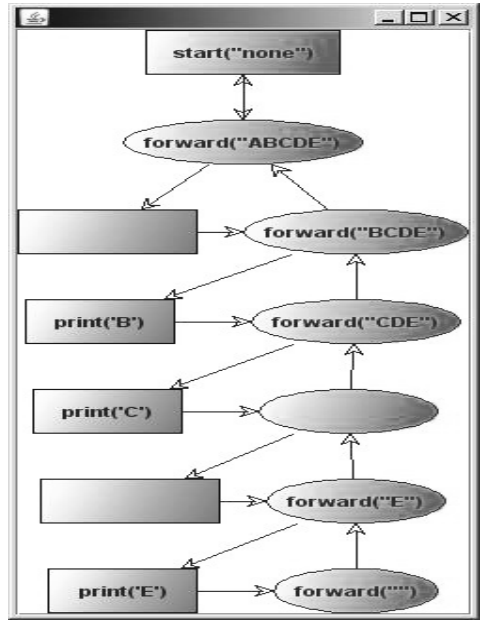

Figure 4. An RGraph for Forward Printing of list "ABCDE" with 30% of labels missing

Comparison between an RGraph and a Recursion Tree

This section compares an RGraph and a recursion tree. As stated in the Introduction, the major difference between an RGraph and a recursion tree is that a recursion tree exhibits an abstract concept; whereas an RGraph shows a detailed invocation sequence.

To illustrate the difference, we use the recursion tree in Figure 5 of chapter 17 in (Horstmann, 2002) as a comparison example. In this example, the growth of rabbit population is being calculated. The following describes the problem specification.

In a simplified rabbit-growth world, a rabbit, in its first two months of life, does not bear babies. Every month after the first two months, each male and female pair gives birth to exactly one pair of male and female babies. The problem is to find the number of rabbit pairs after n months starting with just one pair of rabbits. Define $rabbit(n)$ as the number of rabbit pairs in n months. The recurrence relation of the problem can be formulated as

$$rabbit(n) = \begin{cases} 1 & if \quad n = 1 \\ 1 & if \quad n = 2 \\ rabbit(n-1) + rabbit(n-2) & if \quad n > 2 \end{cases}$$

A recursion tree for the recurrence relation in the above equation is illustrated in Figure 5 for the case $n = 5$. An RGraph for the same recurrence relation is shown in Figure 6.

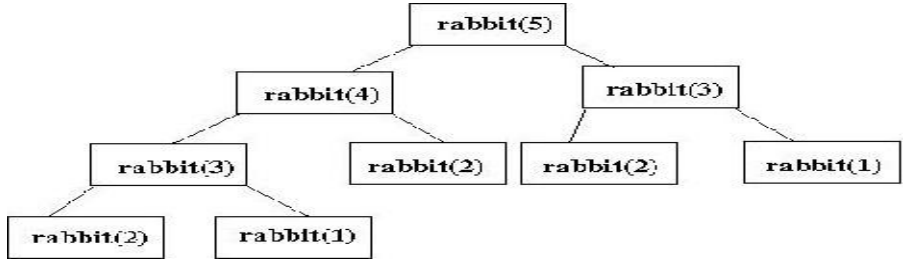

Figure 5. A Recursion Tree for counting rabbit growth

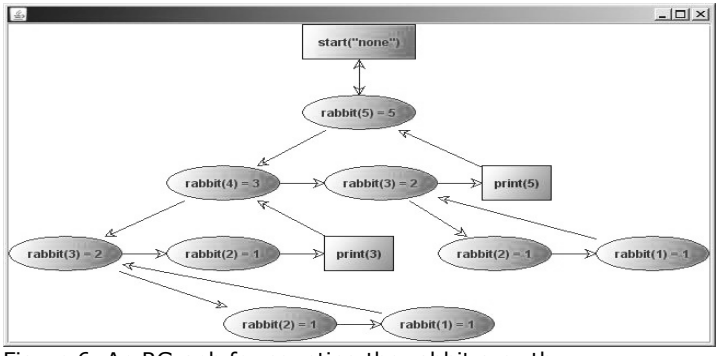

Figure 6. An RGraph for counting the rabbit growth

Comparing Figures 5 and 6, an RGraph explicitly shows the calling and returning sequence by following the direction of the edges; whereas Figure 5 only shows how the recursion progresses to degenerated cases.

Experimental Result in Student Learning

The RGraph software was implemented in 2009. In the past, before RGraph software was available, students would hand draw RGraphs for recursion problems. In the interest of understanding how the RGraph software tool impacts student learning, in 2009 fall semester, we conducted RGraph pre- and post- surveys in 3 undergraduate Computer Science courses, ranging from Discrete Mathematics to Programming Languages, with a total of 34 students. Each survey is given 5 questions as listed in Table 1 with a value of 5 (Strongly Agree), 4 (Agree), 3 (Neutral), 2 (Disagree), 1 (Strongly Disagree), and 0 (not applicable). Table 1 shows the average result of the pre- and post-surveys. Prior to using RGraph software, since students do not know what RGraph is, the pre-survey shows that the students are neutral about RGraph. After introducing RGraph software, it can be seen from the survey result that in general, students strongly agree in all questions regarding the use of RGraph software tool.

Table 1. RGraph Pre-Survey and Post-Survey Result

Survey Question	Pre-Survey Average	Post-Survey Average
(A) RGraph can help me trace the flow of recursion	3.65	4.59
(B) RGraph is a visual aid to illustrate the process of recursion	3.85	4.88
(C) Compared to Horstmann's Recursion Tree, RGraph can show the process of recursion more clearly	3.15	4.62
(D) RGraph helps me understand the concept of the recursion	3.44	4.56
(E) Using RGraph, I am more comfortable with the concept of recursion	3.41	4.53

Summary and Conclusion

The concept of recursion is important to many fields of study, especially when many applications rely on computer software for data analysis and prediction. Through years of conveying the concept of recursion to students in Computer Science, the authors have found that learning recursion is nothing more than the old saying: practice makes perfect. However, just as in most learning environments, an adequate learning tool is the key to success. Our invention of RGraph makes the concept of recursion illustratable and traceable, thereby allowing flexibility in demonstrations and focused pedagogical interactions on the part of students.

The model provided in this paper is reflective of one of the goals of the scholarship of teaching and learning (SoTL), in which "the faculty frame and systemically investigate questions related to student learning—the conditions under which it occurs, what it looks like, how to deepen it" (Hutchings & Shulman, 1999). In this paper, we investigate how a visualization technique helps student learning in Computer Science. Our example of pedagogical research can be generalized to other fields of study. Our experimental result shows that an RGraph is a valuable learning and teaching tool.

References

AP Central - Teaching Recursion (n.d.) http://apcentral.collegeboard.com/apc/members/courses/teacherscorner/45406.htm

Bruce, K., Danyluk, A., & Murtagh, T. (2005). Why structural recursion should be taught before arrays in CS1. ACM SIGCSE.

Bruff, D. (n.d.) The Scholarship of Teaching and Learning (SoTL.) Vanderbilt Center for Teaching. http://www.vanderbilt.edu/cft/resources/teaching_resources/reflecting/sotl.htm#what3

Cormen, T., Leiserson, C. Riverst, R, & Stein, C. (2009). *Introduction to Algorithms*. Boston: The MIT Press.

Dann, W., Cooper, S., & Pausch, R. (2000). Using visualization to teach novices recursion. Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education, Canterbury, England, pp. 109-112.

Davidson, A. (n.d.) Eight Queens Java Applet. http://cpaz.ca/aaron/SCS/queens/

Gal-Ezer, J. & Harel, D. (1998). What (else) should CS educators know? Communications of the ACM 41, 9, pp. 77-84.

Haug, F. (n.d.) Relevant algorithm animations/visualizations (in Java). Chapter 5. Recursion. http://www.ansatt.hig.no/frodeh/algmet/animate.html

Horstmann, C. (2002). *Big Java*. John Wiley & Sons, Inc.

Hsin, W.-J. (2008). Teaching recursion using recursion graphs. In the conference proceeding of Consortium of Computing Sciences in Colleges. April.

Hundhausen, C. Douglas, S., & Stasko, J. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3), 259-290. June.

Hutchings, P. & Shulman, L.S. (1999). The scholarship of teaching: New elaborations, new developments. Originally published in the September/October 1999 issue of Change. http://www.carnegiefoundation.org/elibrary/docs/sotl1999.htm

JHAVEPOP. (n.d.) Linked list manipulations using JHAVEPOP. http://jhave.org/jhavepop/java/exercises.html

McCracken, D. (1987). Ruminations on computer science curricula. *Communications of the ACM, 20*(1) 3-5.

McHugh, J. (n.d.) The animation of recursion. http://www.animatedrecursion.com/intro/introduction.html.

Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M. (2004). Visualizing programs with Jeliot 3. Proceedings of the International Working Conference on Advanced Visual Interfaces AVI 2004, Gallipoli (Lecce), Italy.

National Institute of Standards and Technology. (2005). Recursion tree. http://www.itl.nist.gov/div897/sqg/dads/HTML/recursionTree.html.

Recursion Tree. (1997). http://www.cs.duke.edu/courses/fall97/cps130/lectures/lect04/node24.html.

Recursion Tree. (2007). http://homepages.ius.edu/rwisman/C455/html/notes/Chapter4/RecursionTree.html.

Stern, L. & Naish, L. (2002). Animating Recursive Algorithms. http://imej.wfu.edu/articles/2002/2/02/index.asp

Turbak, L. (2001). Recurrence in CS231: Algorithms. http://cs.wellesley.edu/~cs231/fall01/recurrences.pdf.

Wu, C., Dale, N., & Bethel, L. (1998). Conceptual Models and Cognitive Learning Styles in Teaching Recursion. ACM SIGCSE.

Leonardo Sa is currently a senior in the Department of Computer Science, Information Systems, and Mathematics at Park University. His interest is in the area of computer programming and networking. He has many years of working experience as a program analyst prior to coming to Park University.

Wen-Jung Hsin received her interdisciplinary PhD in Telecommunications and Computer Science at the University of Missouri - Kansas City. She is currently a professor in the Department of Computer Science, Information Systems, and Mathematics at Park University. Her teaching and research interests are in the areas of Computer Science education, computer networking, and network security.